KUIS Technical Report

KUIS-95-0006

A Proposal of Self-Cleanup Cache

Shin-ichiro MORI, Masahiro GOSHIMA,
Hiroshi NAKASHIMA and Shinji TOMITA

Dep. of Information Science
Faculty of Engineering, Kyoto University
Sakyo-ku, Kyoto, 606-01 Japan

E-mail: moris@kuis.kyoto-u.ac.jp

March 1995

A Proposal of Self-Cleanup Cache

Shin-ichiro MORI, Masahiro GOSHIMA,
Hiroshi NAKASHIMA and Shinji TOMITA

Dep. of Information Science
Faculty of Engineering, Kyoto University
Sakyo-ku, Kyoto, 606-01 Japan
E-mail: moris@kuis.kyoto-u.ac.jp

Abstract

An improved Write-Back(WB) cache which we call *Self-Cleanup Cache:SCC* is proposed in this paper. The SCC aggressively enforces write-backs by itself to update memories. In this way, the SCC keeps itself as clean as possible, while preserving the write merge effect of WB cache.

The SCC is originally designed to efficiently implement the invalidation based Compiler Aided Cache Coherence(CACC) scheme on the multiprocessor system which configured with WB caches. However, even for the systems which adopt directory based cache coherence scheme, the SCC has a good effect on decreasing read latencies.

Two primal effects of the Self-Cleanup Cache, 1) distribution of write-back transactions over a period of execution time and 2) the aggressive write-back, are examined by simulation upon two cache system models: 1)the system with the SCC and the selective invalidation scheme which is one of the CACC schemes, and 2)the other system with the SCC and the full-map directory scheme, respectively.

The simulation results show that 1) the effect of the write-back distribution appears as the decrease of the stall time at synchronizations, and thus improves the performance of the selective invalidation scheme, and 2) the aggressive write-back decreases the read stall time though it may increase unnecessary write-back transactions. It is also confirmed that, for an well-optimized program, the CACC scheme gains an advantage over the directory scheme.

1 Introduction

Rapid increase of the processor speed nowadays has been leading to a wide gap between processor speed and memory speed. In order to exploit the processor's maximum performance, it is required to narrow this gap. The cache memory plays an important role for this purpose.

On the other hand, in the large scale shared memory multiprocessors, the more the size of the system scales, the more the memory access latencies increase. The latency hiding with cache memory hence becomes much more important. Furthermore, toward the massively parallel processor, it is indispensable to develop techniques to reduce the latency itself as well as the latency hiding technique.

Most of the researches on the cache memory so far have been directed toward latency hiding techniques like that to reduce the miss ratio from the standpoint of cache organization[22], and it has been less studied how to reduce the latency itself. The *write-buffer* approach for the Write-Through(WT) cache to reduce the write latency is one of those that have been well studied by now. Together with some appropriate memory ordering model [13, 14, 16], the *write buffer* helps to reduce the write latency even for the Write-Back(WB) cache.

However, the latency which dominates the processor performance is not the write latency but the read latency, because the processor essentially can not proceed its program execution, on the read miss, until the required data is supplied. Although both hardware-based[24] and software-based[5] prefetching techniques have been proposed to decrease this read latency, they don't work well if the latency is too large.

In the Write-Back cache under the directory based cache coherence scheme particularly, if the memory does not contain up-to-date data, the read latency becomes 1.5 to 2 times longer than that the case where the memory has up-to-date data. This is because of the write-back overhead inherent in the write-back cache.

In order to decrease such read latency in the write back cache, we propose a new cache system with self-cleanup control, which we refer to as Self-Cleanup Cache(SCC).

Although the SCC is originated in Write-Back cache, by means of self-cleanup control, the SCC can keep itself as clean as possible, like the Write-Through cache, while maintaining the write merge effect of the Write-Back cache. In order to realize this, we introduce a sort of working set to manage the dirty line inside the cache. By means of enforcing the write-back of dirty line when it is forced out from the working set, the SCC reduces the read latency for the other processors. In some sense, it can be said that this is a proposal of a new write policy which has the characteristics intermediate between the write-back and the write-through policies.

The next section of this paper describes the background of this research, including the motivation of this proposal. In section 3, the concept of the Self-Cleanup Cache is given with an implementation example. In section 4 and 5, overview of simulation environment and

simulation results are presented to demonstrate effectiveness of the SCC.

2 Related Works

2.1 Compiler Aided Cache Coherence Control

Directory based cache coherence schemes become a conventional approach to the cache coherence problem in recent distributed shared memory machines. With the directory scheme, however, the run-time overhead to enforce the coherence increases rapidly and becomes unacceptable as the system scales up. This fact has motivated to devise software assisted cache coherence scheme such as the cacheability marking scheme[4] and self-invalidation scheme [9, 10, 19, 11, 8, 15, 21, 17] , which is referred to as Compiler Aided Cache Coherence(CACC) schemes in this paper.

Most of the CACC schemes rely on self-invalidation of the stale data by each processor and require the main memory to have up-to-date data at certain points, which we call *synchronization points*, of program execution on each processor [1]. Therefore they don't suit well with the conventional write-back cache, because the up-to-date data can be contained not in memory but in cache.

The Self-Cleanup Cache (SCC) has been originally designed to implement the CACC schemes on the system with the Write-Back cache.

In order to apply the CACC scheme to the system with Write-Back cache, at the synchronization points, all up-to-date data have to be written-back to guarantee that memories contain the latest data with respect to the processor, prior to the invalidation. Although several invalidation techniques [9, 10, 19] can be applied to this situation, the efficient means for the write-back are not developed yet. In order to develop the efficient means, we must consider the the following two issues associated with the write-back.

- Which lines should be written back?
  Without any cooperation from hardware coherence mechanisms like directory or snooper, there is no way to know immediately which cache lines are in the dirty state. The only way which we can do to find the dirty lines is the exhaustive search over the cache. Of course, there is yet another approach which relies on the compiler to explicitly insert the codes to force the write-back one by one, based on the complete static analysis taking the cache into consideration. Even in the later case, it is still not so efficient without any assistance from the hardware. Thus it is required to develop the hardware to cope with this issue.

- When should the cache write dirty lines back to memory ?

  If processors start writing all the dirty lines back to memory when they reach to any synchronization point, the network traffic increases rapidly, particularly in DOALL type parallel processing. This may cause the network saturation which brings incredible performance degradation.

  In order to prevent such situations, we should develop some techniques to scatter the write-back traffics over the time.

One solution to these questions is the Self-Cleanup Cache (SCC) proposed in this paper.

Requirement 1 Keep track of the addresses where a processor made write accesses.

Requirement 2 Write-backs of lines at appropriate points distributed over a period during program execution.

2.2 Latency Tolerance

Latency Tolerance is very important issue in the development of memory system. Weaker form of memory ordering models [13, 14, 16] are techniques to hide memory access latency. By means of

1. overlapping of processor execution and memory accesses,
2. buffering of memory accesses, and
3. concurrent memory accesses which are essentially no need to be serialized,

the weak form of memory ordering models provide the following effects.

- Latency hiding by means of background processing of coherence enforcement allowing the temporal inconsistency, and
- Throughput improvement through the pipelining of write accesses.

However, in spite of these advantages of the ordering models, in the system with Write-Back cache, there exist some situations where any ordering model do not either help to hide nor reduce the latency. These situations are due to the issue that memory may not hold the latest data.

Figure 1 demonstrates such situation, where a processor Pi causes a read miss hit on a line A which is previously cached in another processor Pj in the dirty state. In this case, the read access experiences the latency of roughly 2 times[2] longer than that for the case where the memory has up-to-date data.

However, if the system has ability to satisfy the following requirement, the system will suffer such long read latency less than the system without this ability.

[1] Based on the static analysis by a compiler, instructions to invalidate the potentially stale cache lines are inserted at some kind of synchronization points, such as parallel loop boundaries or the entrances and the exits of the critical sections. When the program is executed, the processor will issue instructions to invalidate cache lines and the invalidated copies will be updated during subsequent read misses to these cache line.

[2] If the cache has a facility of direct cache-to-cache transfer [18], it will be 1.5 times.

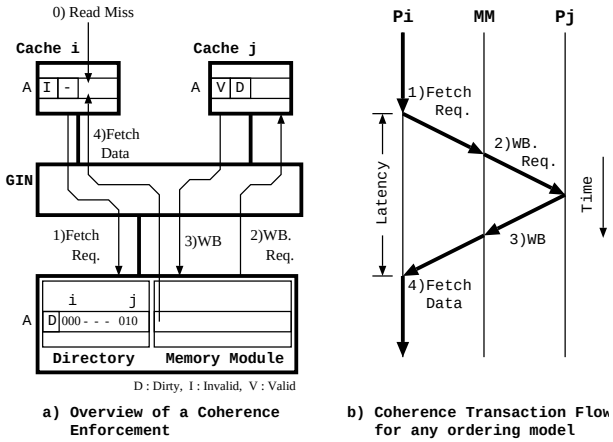

Figure 1: Read Latency on Write-Back Cache.

Requirement 3 Dirty lines which are not likely to be accessed any more are written back to memories as soon as possible.

This is what we exactly do in the Self-Cleanup Cache, so that it could hide the latency of write-back and could reduce the latency of cache refill, in cooperation with the buffering and pipelining.

3 Self-Cleanup Cache

3.1 Overview of the Self-Cleanup Cache

The Self-Cleanup Cache(SCC) has been proposed

- to realize fast and efficient CACC control over the system configured with WB cache, and
- to reduce the write-back overhead inherent to the write-back cache, particularly for the system which adopts a sort of loose memory ordering model.

As shown in Figure 2, SCC is composed of two units: a conventional write-back cache unit and an additional Self-Cleanup Control unit.

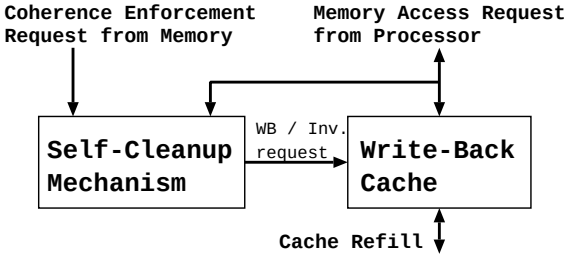

Figure 2: Organization of SCC

Self-Cleanup Control unit keeps track of the addresses of the dirty lines in its partner cache and forces the cache unit to write-back by informing when and which line should be written back. By means of requesting the write-back of a dirty line at appropriate time, those requirements described in section 2 are satisfied.

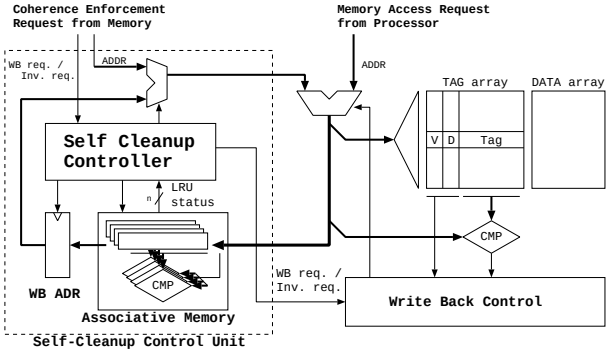

Figure 3: An implementation of the SCC with LRU replacement policy.

In this way, the Self-Cleanup Cache holds the feature of write aggregation on Write-Back cache, while keeping both cache and memory as clean as possible. And consequently SCC can reduce the memory access latency as we have discussed in the previous section. In some sense, it can be thought that this is a proposal on write policy which is intermediate between the write-back and the write-through.

3.2 Self-Cleanup Strategies

The Self-Cleanup cache takes the advantage of the temporal and spatial localities of memory access [22] to make decisions on when and which line should be written-back. Based on these localities, a sort of working set of dirty lines is defined and appropriately managed by self-cleanup control. The dirty line which is forced out from this working set becomes next candidate of write-back.

The definitions of the working set and its management strategy characterize the Self-Cleanup Cache. This means that there may exist various implementations of the SCC according to those self-cleanup strategies.

In a practical implementation, however, the self-cleanup mechanism should not be a time critical unit in the cache design. Thus practical self-cleanup strategy will be something like as follows.

Working set : N dirty lines which are not private to a particular processor.

Management strategy : First-In-First-Out and Least-Recently-Used replacement policies.

We show such an example in the next section.

3.3 An Implementation Example of the SCC with LRU Replacement Policy

We show an SCC implementation which manages the working set in LRU replacement policy. For the simplicity of explanation, we assume all accesses are to the shared data.

3.3.1 Hardware Organization

Figure 3 shows the organization of the SCC with LRU management. Basic components of SCC are a conventional write-back cache unit and an additional Self-Cleanup Control unit. The Self-Cleanup Control unit consists of the following 4 subunits.

- Associative Memory : keeps track of addresses of dirty lines in a cache.
- Write-Back ADdress Register (WB ADR) : holds the next candidate address to be written back for self-cleanup.
- Address Multiplexor : switches Write-Back Request (WBreq) from self-cleanup control and Cache Coherence enforcement Request (CCreq: Invalidation req. and write-back req.) from memory.
- Self-Cleanup controller : determines the conditions to register/remove the address to/from the associative memory according to the self-cleanup strategies. The address removed from the associative memory is first stored to the WB ADR as the next candidate of write-back.

3.3.2 Fundamental Operations

The fundamental SCC operations observed by processor or memory are completely the same as those of conventional write-back cache. The only difference observable from outside is the amount of write-back transactions.

Write-back transactions in the SCC are initiated in the following three cases.

1. When a line replacement is required at cache miss and a dirty line is chosen as the replacement victime, a Write-Back for the line replacement occurs.
2. When a memory requests the cache to write-back a dirty line, a Write-Back to keep coherence occurs.
3. When the self-cleanup controller requests the SCC itself to initiate a write-back, a Write-Back to keep itself clean occurs.

The third situation forces the SCC to initiate more write-back transactions than conventional write-back caches. For the sake of clarity, we refer to the write-back request initiated by self-cleanup controller as *self-cleanup request* in the following context.

In this implementation of the SCC, when a write miss[3] occurs, the self-cleanup controller requests the write-back if all entries of associative memory have been occupied.

In the following, we show the transaction sequence of the SCC when a write miss occurs, provided that the priority of write-back requests from memories or the SCC is higher than the priority of memory access requests from processor, and that no preemption is allowed.

- If there is available entry in the associative memory,

  register the address of the line at where the processor causes a miss to the associative memory and update the LRU history of each entries, then proceed with the ordinary miss hit transaction.

- If there is no available entry in the associative memory,

  1. Remove the address marked with Least-Recently-Used and store it to the WB ADR. And then, by asserting the WB req signal, inform the Write-Back cache units that the SCC has an address of line to be written back.
  2. Register the address of the line where the processor causes a miss to the associative memory and update the LRU history.
  3. Proceed the ordinary miss hit transaction for the missed address.
  4. Proceed the write-back transaction for the line indicated by the WB ADR. Note that the line needs not to be invalidated.
  5. If there are some requests from memory waiting responces from the cache, respond to all such requests.
  6. Negate the WB req signal and start accepting the request from the processor.

Whenever a write hit occurs in its partner cache, LRU history of the associative memory is updated.

As we have mentioned earlier, there are several alternatives in implementing the SCC even with LRU replacement policy. Some of these alternatives are shown below.

- Ignorance of *self-cleanup requests*

  As we mentioned, the SCC works the same as the conventional write-back cache with an exception that it initiates write-back transactions for self-cleaning. And if the write-back cache unit ignores all the self-cleanup requests, the SCC is completely the same as the conventional write-back cache.

  Thus there is an option that the write-back cache unit ignores some of the self-cleanup requests, though this is not allowed if the SCC is used to realize the Compiler Aided Cache Coherence (CACC) scheme in the system with write back cache.

  For instance, this option may be used to prevent the network from being saturated. In this context, the SCC abandons the self-cleanup requests which are always not in a hurry when the network is too busy.

- Start requesting the self cleanup when more than half of the associative memory become occupied.

  In this case, the SCC may postpone the write back for self-cleanup as long as there are some available entries in the associative memory. For example, with this option, the SCC may wait initiating the self-cleanup request while the network is busy.

[3] It may be because of either tag mismatch or write hit to a clean line.

This option will hopefully help to distribute the network traffic and function as a sort of load balancing mechanism. It will provide much better effect if the write-buffer is configured with the SCC to allow memory access pipelining[13].

- The dirty line flush function
  This is the indispensable function to efficiently implement the CACC control over the system with write-back cache.

  The invalidation based CACC control [9, 10, 19] requires the memories to contain up-to-date data at certain points during program execution. In order to realize this, each write-back cache has to force the write-backs corresponding to all dirty lines in each cache at these points.

  Using the dirty line flush function of the SCC, these write-backs can be done in constant time in proportion to the associative memory size, since all the dirty line addresses remain in the cache are registered in the memory, so that we don't need any exhaustive search on each cache.

3.4 Comparison with the Write Buffer Approach

The Self-Cleanup Cache with LRU replacement policy provides almost equivalent functions to the write-through cache with write-buffer using the write merging strategy[8, 28]. Here we make a brief comparison between the SCC and the Write-Through Cache with the write-buffer.

The advantage of the SCC as compared with the write-buffer approach is that it doesn't need the additional data buffer other than the data-array in the cache. In addition, in order for the write buffer approach to be equivalent to the SCC, the write-buffer has to adopt the write merging strategy which requires further hardware as much as small write back cache. Taking these facts into consideration, we could state that the Self-Cleanup Cache has an advantage in cost-performance.

At this point, one may claim that there must be some possibility of the contention on data array between processor's access requests and write-back requests from self-cleanup controller. Fortunately, as we showed in the previous sections, the self-cleanup request is motivated in processor's write miss in principle, and at this time processor therefore has been in stall waiting for the completion of the write miss transaction. It may thus proceed the write-back transaction for self-cleanup in behind the write miss transaction including the ownership acquirement, so that SCC is able to be free from the contention.

One may claim also, however, that there exist some ordering models where processor doesn't stall on write miss on clean line. In this case, we may use some tricky techniques which postpone the write-back for self-cleanup until the next read/write miss that causes processor stall[29].

Furthermore, even if the associative memory of *SCM* is full when write miss occurs, the SCC does not bring the processor stall unless the SCC is used with CACC scheme, whereas in the write-buffer approach processor always stalls if the write-buffer is full when write occurs. The reason why is that, in this context, the SCC may work as write-back cache which may buffer the up-to-date data inside its data array, thereby the SCC does not need to keep the complete address trace and also it may temporarily ignore the write-back for self-cleaning as we mentioned in the previous section.

4 Methodology of Performance Evaluation

4.1 Goal of the Performance Evaluation

In order to demonstrate the practical effect of Self-Cleanup Cache, we did some performance evaluations by simulation. The Goal of the evaluations is to confirm the following facts.

1. Self-Invalidation based CACC schemes can work efficiently with the assistance of the Self-Cleanup cache with write-back write-policy.
2. Read stall time can be reduced as an effect of aggressive write-back initiated by Self-Cleanup mechanism.

4.2 Simulation Models

In our simulation works, we have examined two cache system models described below. The Self-Cleanup mechanism with LRU replacement policy is assumed throughout the simulation works. Private data is excluded from the object of aggressive Write-Back.

SISC model is a system which adopts **S**elf **I**nvalidation based compiler assisted cache coherence scheme in combination with **S**elf-**C**leanup Cache. In the *SISC* model, whenever a processor issues a synchronization instruction, all dirty lines in the processor's cache are written back to memory and then all cache lines except for the line which contain private data is invalidated, before the instruction is completed. We assume this invalidation is executed in one machine cycle. The Self-Invalidation scheme used in this model is equivalent to the Fast Selective Invalidation scheme[9] under the condition where all accesses to *shared data* are marked with *memory_read* and the rest (access to *private data*) are marked with *cache_read*.

DISC model is a system which adopts **DI**rectory based cache coherence scheme with the assistance of **SCC**. We assumed full-map directory scheme and Illinois protocol (4 states: MOdified, Exclusive Clean, Shared, Invalid) as its coherence protocol.

Execution time and its details[4] (time spent in computation, read stall time, write stall time, and time spent

[4] Details of execution time is measured on a processor in which the sum of read and write stall times is maximum of all at LRU entry of 4.

waiting the completion of synchronization), and total number of messages exchanged during a program execution and its details (Read Requests and Write Requests issued by caches, and Write-Back Requests and Invalidate Requests issued by memories) are measured under all combination of the following three parameters:

1. Number of LRU entry(4,8,16,32,WOSCM[5]) ,
2. Cache size(1MB,256KB,64KB,32KB[6])
3. Network latency(10clk, 20clk/hop).

In case of the *SISC* model without *SCM* , the exhaustive search of dirty lines is made on each cache at synchronization points. However, the simulator ignores the time spent in this search, assuming that it can be overlapped with the time spent in doing write-backs.

In the simulation, we use 2 benchmarks, SOR and FFT, which have simple memory access pattern and another 2 benchmarks, WATER and MP3D from SPLASH[23], as examples of practical application. Note that only SOR and FFT are used in the simulation of *SISC* model.

4.3 Overview of Simulator

We use an execution driven simulator which consists of processor[7], cache, network and memory modules. The simulator assumes the system configuration shown in Figure 4 with configuration parameters shown in Table 1. And also we did the following assumption:

- Number of processors is 32,
- Direct-mapped cache with write-back write-policy and write-allocate on write-miss policy,
- line size of 32 bytes,
- weak ordering model,
- a torus network with network contentions is simulated,
- at most 4 outgoing write requests (including ownership requests) and 1 read requests are proceeded concurrently, and
- no write buffer exists between cache and network.

5 Simulation Result

5.1 Summary of Results

Figures 5 to 14 show the simulation results. The x-axis of the figure indicates the number of LRU entries of Self-Cleanup Mechanism(SCM). WOSCM means the configuration without SCM.

The summary of the results is shown below followed by detailed analysis in the following 4 sections for each benchmark program.

[5]Without *SCM*

[6]only for FFT and SOR

[7]Processor module simulates a SPARC microprocessor with some extension of primitives, like PARALLEL START/END instructions, for parallel execution. This module had developed with Spa package V1.0 developed in Adelaide university, Australia for reference.

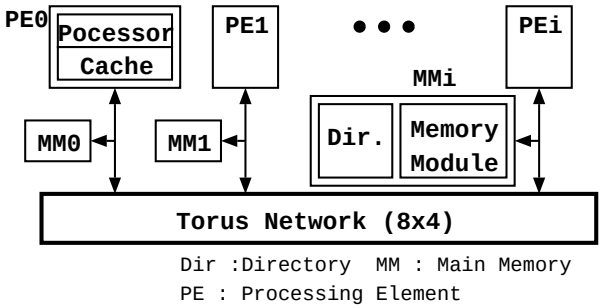

Figure 4: System configuration

Table 1: System configuration parameters

Processor	Execution time of any one instruction	1[cycle]
Cache	line access time	4[cycle]
	word access time	1[cycle]
	cache tag access time	1[cycle]
Network	without Data(1hop)	10[cycle]
	with 32Byte Data(1hop)	14[cycle]
Memory	line access time	10[cycle]
	word access time	4[cycle]
	directory access time	4[cycle]

1. With respect to the *SISC* model, we could confirm the effect of the write-back distribution on the decrease of the waiting time at synchronization points. As a result, the speedups of the configurations with *SCM* over the configuration without *SCM* are almost 10% for SOR. For FFT, execution times are similar to each other configurations because the unnecessary write-back transactions increase the write stall time.

   On the other hand, in spite of invalidating all shared data at synchronization point, the execution time of SOR and FFT are 25% and 2% over the corresponding execution times on *DISC* model, except for the configuration without *SCM* . Although 25% increase of execution time for SOR is not acceptable, this performance degradation is due to the over invalidation rather than the unnecessary write-backs. Furthermore, if the optimization to distinguish private data from shared data is enforced on the program, *SISC* model becomes faster than the *DISC* model

2. With respect to the *DISC* model, we could confirm the reduction of read stall time as an effect of aggressive write-back from cache. Note that, for SOR and FFT, increase of write stall time is greater than the decrease of read stall time. And thus the execution time of the system with the *SCM* is 3% longer, at worst, than the system without the *SCM* .

   We also confirmed that the *SCM* never increases the cache miss from the fact that the number of read request messages is constant regardless of the configuration of *SCM* .

3. In both models, the number of LRU entries of *SCM* has some effect on the number of messages but not on the execution time. The only exception is MP3D where the number of LRU entries has some effect on the execution time.

4. To the extent of what our simulation study covers, the difference of network delay has no effect on the behavior of the system other than the proportional increase of execution time. Therefore, we consider the simulation result only for the network delay of 10 clock in the following discussion.

5.2 SOR

5.2.1 Characteristics of the problem

SOR is a Partial Differential Equation solver using Even-Odd method. A 256×256 grid is divided into 32 subgrids of 32×64 and the subgrids are assigned to processors one by one. In our simulation, we have measured the time and number of messages for the first 2 time steps excluding the sequential execution part for initialization. Since all subgrids are treated as shared data, performance degradation caused by unnecessary write-back transactions may occur. This is an example which exposes the drawback of Self-Cleanup Cache.

5.2.2 Analysis of the results

In the *SISC* model, the number of messages and its details are almost the same for all configurations. The differences in execution time are caused primarily by the differences in waiting time at synchronization points. All dirty lines in the cache must be written back to memory at synchronization points but there is no computation to overlap with write-backs at these points. Thus the waiting time is directly proportional to the number of dirty lines to be written back. As a result, the configurations with *SCM* become faster than the configuration without *SCM* .

In the system configured with *SCM* , write-back messages initiated by *SCM* increase significantly in *DISC* model. However, it does not affects the execution time too much because these write-backs overlap with the computation. Although read-stall time decreases as an effect of aggressive write-backs, it is slightly less than increase of the write-stall time, then the total execution time increases.

As we mentioned before, execution time on *SISC* model is roughly 1.25 times longer than on *DISC* model. This performance degradation is due to the over invalidation at synchronization points rather than the unnecessary write-backs. Hopefully, both these causes can be alleviated by some program optimization such that unshared data is appropriately declared as private data.

Figure 7 and Figure 8 show the improved execution time and number of messages after the optimization where just only the outmost part of the subgrid is declared as shared. By applying this optimization to original SOR program, a minimum number of messages which are essentially required are exchanged in *SISC* model, while additional messages for coherence

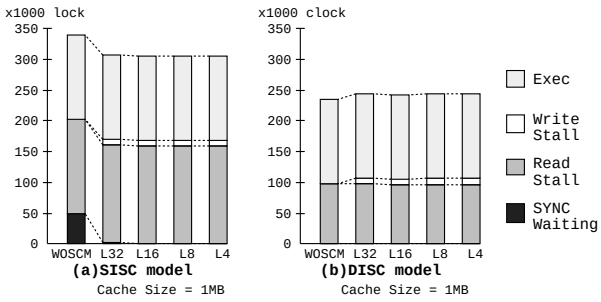

Figure 5: SOR : Execution time

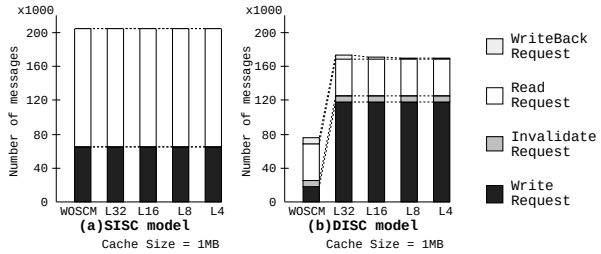

Figure 6: SOR : Number of messages

enforcement, like ownership/invalidation request and its corresponding acknowledgement, are required in *DISC* model. As a result, the performance of the *SISC* model overcomes the *DISC* model.

The effect of the number of LRU entries in *SCM* slightly appears on the write-stall time and synchronization waiting time in *SISC* model, while in *DISC* model the effect appears on the number of Write-back request and on its corresponding write-stall time.

5.3 FFT

5.3.1 Characteristics of the problem

FFT is a parallel 2048-point FFT of complex data without data rearrangement during computation. 2048 input data are distributed among processors in block distribution manner. All the weight constants are previously calculated and delivered to each processor as private data. Parallel Computation is based on the data owner computes rule. During the execution of logN steps, each processor writes to two 16 consecutive cache lines though only half of them are written at each stage of FFT computation.

5.3.2 Analysis of the results

In case of LRU entries of 32 in the *DISC* model, the system behaves completely the same as the system witout *SCM* , since all working set can be kept inside the LRU entries of *SCM* .

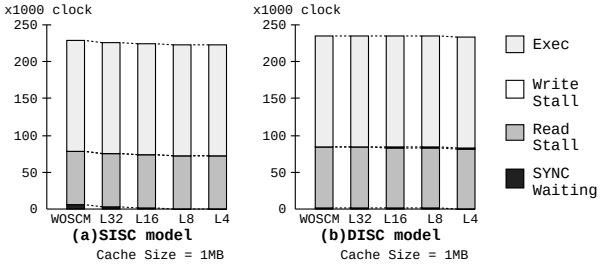

Figure 7: SOR : Execution time (after optimization)

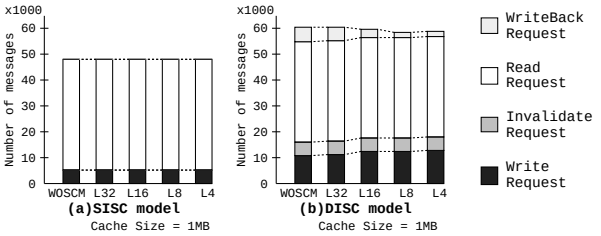

Figure 8: SOR : Number of messages (after optimization)

Although read stall time slightly decreases, increase of unnecessary wait at synchronization point leads at worst 2% increase of execution time as a result.

After the computation reaches at a certain step, no interprocessor communication occurs. And then no more synchronization is required until the end of computation, which makes the working area of each processor to be 32 cache lines. This is the reason why the configuration with LRU entries of 32 and without *SCM* provide faster execution time than the other configurations with less LRU entries.

5.4 WATER

5.4.1 Characteristics of the problem

WATER is an N-body molecular dynamics application that evaluates forces and potentials in a system of water molecules in the liquid state. Gear's sixth-order predictor-corrector method is used to solve the Newtonian equations of motion for molecules. In our simulation, problem size of 64 molecules is used and the computation is performed over 2 time-steps. Any measurement is done for only the parallel execution part.

The data set size of 600Byte is required to store the information for each molecule. Parallelism is implemented by assigning every processor a set of molecules.

5.4.2 Analysis of the results

We could confirm the reduction of read stall time to be 10% to 20% by configuring with the *SCM* . However, the increase of write stall time and synchronization waiting time make the reduction of total memory access time

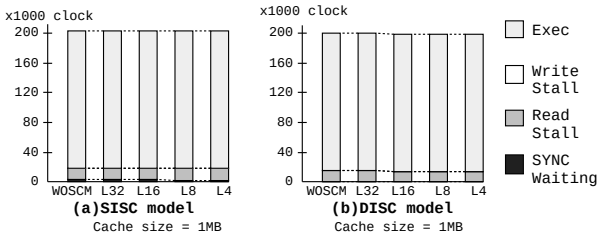

Figure 9: FFT : Execution time

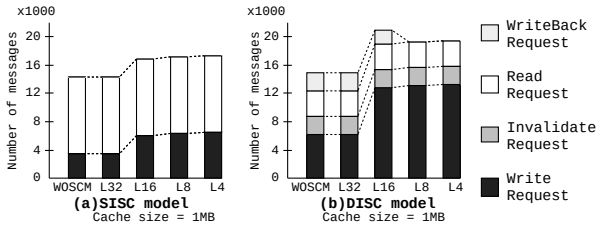

Figure 10: FFT : Number of messages

to be 5%. Furthermore, the final speed up obtained is at most 2% since the computation time is the dominant factor of the execution time.

5.5 MP3D

5.5.1 Characteristics of the problem

MP3D is a program of rarefield hypersonic flow simulation using Monte Carlo method. In our simulation, we assume an active space of rectangular tunnel, composed of $16 \times 16 \times 8$ unit-sized cells, with openings at each end and reflecting walls on their remaining sides, and a single flat sheet placed in the space. Actions of 4096 particles (diatomic molecules) are simulated and the computation is performed over 2 time-steps. Any measurement is done for only the parallel execution part.

Each particle requires 36 bytes to store its state information and each cell requires 40 bytes to store its properties. These data are statically allocated among processors and each processor computes the actions of a certain number of particles statically assigned to itself.

5.5.2 Analysis of the results

When the cache size is 1MB, if the system configured with *SCM* , it can be obtained at most 45% improvement on execution time as compared with the execution time for the configuration without *SCM* .

Since the memory access time is the dominant factor of execution time, not like WATER, the reduction of read stall time as an effect of aggressive write-back directly reduces the execution time.

When the cache size is 64KB, however, we could not observe any advantage of aggressive write-back. This is because the probability that the memory contains up-

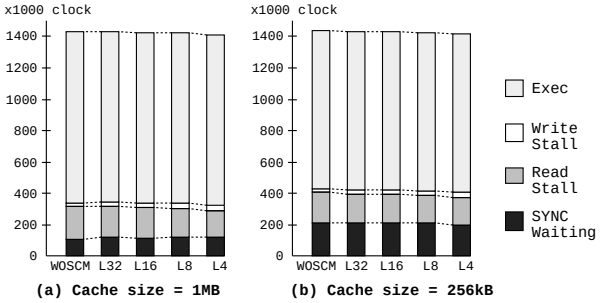

Figure 11: WATER : Execution time

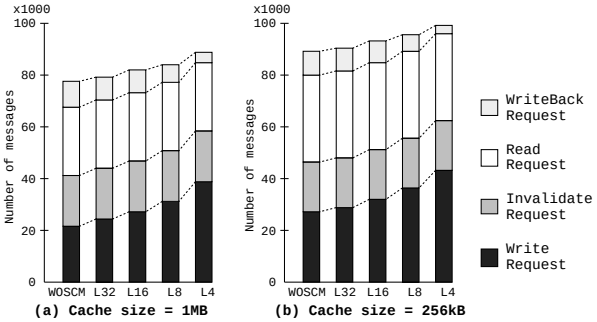

Figure 12: WATER : Number of messages

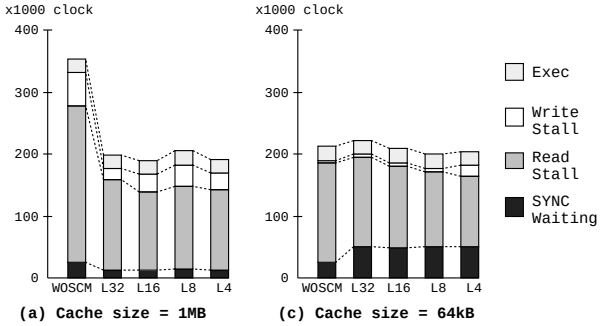

Figure 13: MP3D : Execution time

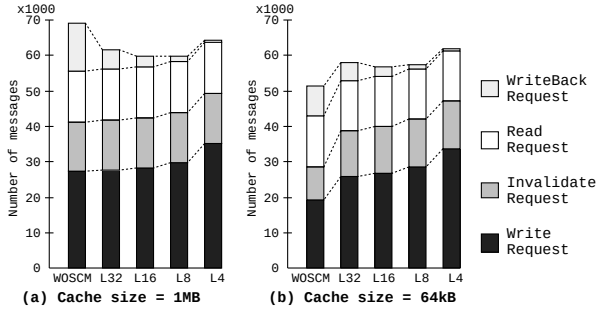

Figure 14: MP3D : Number of messages

to-date data becomes high as a result of frequent cache line replacements which occur regardless of the existence of *SCM* . In addition, if the number of LRU entries is 32, execution time is increased by 4% as compared with the configuration without *SCM* because the write-backs initiated by the *SCM* disturbs the cache line replacements.

6 Conclusion

We have proposed a new cache system which we refer to as Self-Cleanup Cache. The SCC is composed of two units, write-back cache unit and self-cleanup mechanism. The self-cleanup controller keeps track of the address of the dirty lines in the cache and forces the write-back of the dirty lines which are not likely to be accessed by the processor any more. By means of this self-cleanup control, the SCC can keep itself as clean as possible, while maintaining the write merge effect of WB cache.

Two primal effects of the Self-Cleanup Cache, 1) distribution of write-back transactions over a period of execution time and 2) the aggressive write-back, are examined by simulation upon *SISC* model and *DISC* model, respectively. The simulation results show that 1) the effect of the write-back distribution appears as the decrease of the stall time at synchronizations, and thus improves the performance of the selective invalidation scheme, and 2) the aggressive write-back decreases the read stall time, though it may increase unnecessary write-back transactions which will not affect the execution time too much. It is also confirmed that, for an well-optimized program, the CACC scheme gains an advantage over the directory scheme.

Acknowledgment

We should mention that the parallel system simulator used in our simulation study is an modification of a simulator developed by Takeo Hosomi of NEC and Kiyoshi Kamishima of Fuji Xerox while they are students in Kyoto University. We would like to thank them for their assists in the modification.

We would also thank to the colleague of Tomita laboratory for their useful comments on our research and Mr. Yuji Tsushima for his help in summarizing the large amount of simulation data.

This research is partially supported by the grant-in-aids for Scientific Research on Priority Area #04235103, Developmental Scientific Research #06508001, and, Scientific Research (A) #06402059 by the ministry education, science and culture, JAPAN.

References

[1] Sarita V.Adve and Mark D.Hill, "WEAK ORDERING - A NEW DEFINITION AND SOME IMPLICATIONS," *Computer Sciences Technical Report*, #902, Computer Sciences Department, University of Wisconsin-Madison, December 1989.

[2] Archibald, J. and Bear, J. -L, "An Economical Solution to the Cache Coherency Problem," *Proc. 11th Ann. Int'l. Symp. Comput. Architect.*, pp.355–362, June 1984.

[3] Archibald, J. and Bear, J. -L, "Cache Coherence Protocols: Evaluation Using a Multiprocessor Simulation Model," *ACM Trans. on Computer Systems*, Vol.4, No. 4, pp.273–298, Nov. 1986.

[4] Brantley, W.C., McAuliffe, K. P., and Weiss, J., "RP3 Processor-Memory Element," *Proc. 1985 Int'l. Conf. on Parallel Processing*, pp.782–789, Aug. 1985.

[5] Callahan. David, and Kennedy. Ken, Porterfield. Allan, "Software Prefetching," *Proc. ACM ASPLOS IV*, pp.40 – 52, 1991.

[6] Censier. M. Lucie, and Feutrier. Paul, "A New Solution to Cache Coherence Problems in Multicache System," *IEEE Trans. on Computers*, Vol. C-27, No. 12, pp.1112 – 1118, Dec. 1978.

[7] Yung-Chin Chen and Alexander V. Veidenbaum, "An Improved Write Buffer Design for A Write-Through Cache," *CSRD Report No. 1105*, pp.1–17, May 1991.

[8] Yung-Chin Chen and Alexander V. Veidenbaum, "A Software Coherence Scheme with the Assistance of Directories," *CSRD Report No. 1106*, pp.1–22, June 1991.

[9] Cheong, H. and Veidenbaum, A. V., "A Cache Coherence Scheme with Fast-Selective-Invalidation," *Proc. 15th Int'l. Symp. Comput. Architect.*, pp. 299-307, June 1988.

[10] Cheong, H. and Veidenbaum, A. V., "A Version Control Approach to Cache Coherence," *Proc. Int'l Conf. on Supercomputing '89*, pp. 322-330, June 1989.

[11] Cheong, H. and Veidenbaum, A. V., "Compiler-Directed Cache Management in Multiprocessors," *IEEE Computer*, vol.23, no.6, pp. 39-47, June 1990.

[12] David R.Cheriton, Hendrik A.Goosen, and Patrick D.Boyle, "Pradigm:A Highly Scalable Shared-Memory Multicomputer Architecture," *IEEE computer*, pp.33–46, February 1991.

[13] Michel Dubois, Christoph Scheurich, and Faye A. Briggs, "Memory Access Buffering in Multiprocessors," *Proc. 13th Ann. Int'l. Symp. Comput. Architect.*, pp.434–442, 1986.

[14] Michel Dubois, Christoph Scheurich, and Faye A. Briggs, "Synchronization, Coherence, and Event Ordering in Multiprocessors," *IEEE computer*, pp.8–21, February 1988.

[15] Michel Dubois, "Delayed Consistency," *SCALABLE SHARED MEMORY MULTIPROCESSORS*, ed. by Michel Dubois and Shreekant Thakkar, pp.207–218, Kluwer Academic Publisheers, 1991.

[16] Kourosh Gharachorloo, Daniel Lenoski, James Laudon, Phillip Gibbons, Anoop Gupta, and John Hennessy, "Memory Consistency and Event Ordering in Scalable Shared-Memory Multiprocessors," *Proc. 17th Ann. Int'l. Symp. Comput. Architect.*, pp.15–26, May 1990.

[17] T. Hosomi, "A Hardware Cache Coherence Scheme with the assistance of software," *IPSJ technical report of SIG. Arc.*, Vol.93, No.20, pp.117–124, Mar. 1993 (in Japanese).

[18] Lenoski, D., et al. "The Directory-based Cache Coherence Protocol for the DASH Multiprocessor," *Proc. 17th Ann. Int'l. Symp. Comput. Architect.*, pp.148–159, May 1990.

[19] Min.,S.L. and Baer.,J.-L., "A Timestamp-Based Cache Coherence Scheme," *Proc. 1989 Int'l. Conf. on Parallel Processing*, pp.I-23 – I-32, June 1989.

[20] S.Mori, K. Murakami, E. Iwata, A. Fukuda, and S. Tomita, "The Kyushu University Reconfigurable Parallel Processor – Cache Architecture and Cache Coherence Schemes –," *Proc. Int'l Symp. on Shared Memory Multiprocessing*, pp.218–229, April 1991.

[21] S.Mori, et al., "Overview of the ASURA: A Distributed Shared Memory Multiprocessor," *IPSJ technical report of SIG. Arc.*, Vol.92, No.48, pp.41–48, June 1992 (in Japanese).

[22] David A. Patterson and John L. Hennessy, *Computer architecture : A Quantitative Approach*, Morgan Kaufmann Publishers, Inc., 1990.

[23] J.P.Singh, W.D.Weber and A.Gupta:"SPLASH:Stanford Parallel Applications for Shared-Memory", *Tech. Rep. CSL-TR-91-469,Stanford Univ*,Apr.,1991

[24] Alan Jay Smith, "Cache Memories," *ACM Computing Surveys*, Vol.14, No.3, pp.473-530, Sept. 1982.

[25] Stenström, P., "A Survey of Cache Coherence Schemes for Multiprocessor," *IEEE Computer*, vol.23, no.6, pp. 12-24, June 1990.

[26] Josep Torrellas, et al., "Shared Date Placement Optimization to Reduce Multiprocessor Cache Miss Rates," *Proc. 1991 Int'l. Conf. on Parallel Processing*, pp.II-266 – II-270, 1990.

[27] Veidenbaum, A. V., "A Compiler-Assisted Cache Coherence Solution for Multiprocessors," *Proc. 1986 Int'l. Conf. on Parallel Processing*, pp.1029 – 1039, 1986.

[28] "Alpha Architecture Handbook," *DIGITAL EQUIPMENT Co.*, 1992.

[29] "SuperSPARC User's Guide," *TEXAS INSTRUMENTS*, Oct. 1992.